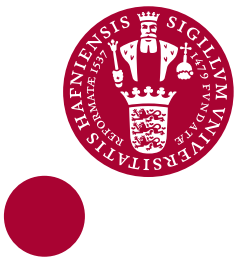

Contracting a planar graph efficiently

Holm, Jacob; Italiano, Giuseppe F.; Karczmarz, Adam; acki, Jakub; Rotenberg, Eva; Sankowski, Piotr

Published in:
25th European Symposium on Algorithms, ESA 2017

DOI:
10.4230/LIPIcs.ESA.2017.50

Publication date:
2017

Document version
Publisher's PDF, also known as Version of record

Contracting a Planar Graph Efficiently*

Jacob Holm[†1], Giuseppe F. Italiano[‡2], Adam Karczmarz[§3], Jakub Łącki[¶4], Eva Rotenberg[5], and Piotr Sankowski[§‖6]

1 **University of Copenhagen, Denmark**
jaho@di.ku.dk

2 **University of Rome Tor Vergata**
giuseppe.italiano@uniroma2.it

3 **University of Warsaw, Poland**
a.karczmarz@mimuw.edu.pl

4 **Google Research, New York**
jlacki@google.com

5 **University of Copenhagen, Denmark**
roden@di.ku.dk

6 **University of Warsaw, Poland**
sank@mimuw.edu.pl

Abstract

We present a data structure that can maintain a simple planar graph under edge contractions in linear total time. The data structure supports adjacency queries and provides access to neighbor lists in $O(1)$ time. Moreover, it can report all the arising self-loops and parallel edges.

By applying the data structure, we can achieve optimal running times for decremental bridge detection, 2-edge connectivity, maximal 3-edge connected components, and the problem of finding a unique perfect matching for a static planar graph. Furthermore, we improve the running times of algorithms for several planar graph problems, including decremental 2-vertex and 3-edge connectivity, and we show that using our data structure in a black-box manner, one obtains conceptually simple optimal algorithms for computing MST and 5-coloring in planar graphs.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases Planar graphs, algorithms, data structures, connectivity, coloring.

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.50

* A full version of the paper is available at https://arxiv.org/abs/1706.10228.

† This research is supported by the Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

‡ Partly supported by the Italian Ministry of Education, University and Research under Project AMANDA (Algorithmics for MAssive and Networked DAta).

§ Supported by the Polish National Science Center grant number 2014/13/B/ST6/01811.

¶ When working on this paper Jakub Łącki was partly supported by the EU FET project MULTIPLEX no. 317532 and the Google Focused Award on "Algorithms for Large-scale Data Analysis" and Polish National Science Center grant number 2014/13/B/ST6/01811. Part of this work was done while Jakub Łącki was visiting the Simons Institute for the Theory of Computing.

‖ The work of P. Sankowski is a part of the project TOTAL that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 677651).

1 Introduction

An edge contraction is one of the fundamental graph operations. Given an undirected graph and an edge e, contracting the edge e consists in removing it from the graph and merging its endpoints. The notion of a contraction has been used to describe a number of prominent graph algorithms, including Edmonds' algorithm for computing maximum matchings [4] or Karger's minimum cut algorithm [11].

Edge contractions are of particular interest in planar graphs, as a number of planar graph properties are easiest described using contractions. For example, it is well-known that a graph is planar precisely when it cannot be transformed into K_5 or $K_{3,3}$ by contracting edges or removing vertices or edges. Moreover, contracting an edge preserves planarity.

While a contraction operation is conceptually very simple, its efficient implementation is challenging. By using standard data structures (e.g. balanced binary trees), one can maintain adjacency lists of a graph in polylogarithmic amortized time. However, in many planar graph algorithms this becomes a bottleneck. As an example, consider the problem of computing a 5-coloring of a planar graph. There exists a very simple algorithm based on contractions [17], but efficient implementations use some more involved planar graph properties [5, 17, 18]. For example, the algorithm by Matula, Shiloach and Tarjan [17] uses the fact that every planar graph has either a vertex of degree at most 4 or a vertex of degree 5 adjacent to at least four vertices each having degree at most 11. Similarly, although there exists a very simple algorithm for computing a MST of a planar graph based on edge contractions, various different methods have been used to implement it efficiently [5, 15, 16].

Our Results. We show a data structure that can efficiently maintain a planar graph subject to edge contractions in $O(n)$ total time, assuming the standard word-RAM model with word size $\Omega(\log n)$. It can report groups of parallel edges and self-loops that emerge. It also supports constant-time adjacency queries and maintains the neighbor lists and degrees explicitly. The data structure can be used as a black-box to implement planar graph algorithms that use contractions. In particular, it can be used to give clean and conceptually simple implementations of the algorithms for computing 5-coloring or MST that do not manipulate the embedding. More importantly, by using our data structure we give improved algorithms for a few problems in planar graphs. In particular, we obtain optimal algorithms for decremental 2-edge-connectivity, finding unique perfect matching, and computing maximal 3-edge-connected subgraphs. We also obtain improved algorithms for decremental 2-vertex and 3-edge connectivity, where the bottleneck in the state-of-the-art algorithms [7] is detecting parallel edges under contractions. For detailed theorem statements, see Sections 3 and 4.

Related work. The problem of detecting self-loops and parallel edges under contractions is implicitly addressed by Giammarresi and Italiano [7] in their work on decremental (edge-, vertex-) connectivity in planar graphs. Their data structure uses $O(n \log^2 n)$ total time.

In their book, Klein and Mozes [12] show that there exists a data structure maintaining a planar graph under edge contractions and deletions and answering adjacency queries in $O(1)$ worst-case time. The update time is $O(\log n)$. This result is based on the work of Brodal and Fagerberg [1], who showed how to maintain a bounded outdegree orientation of a dynamic planar graph so that edge insertions and deletions are supported in $O(\log n)$ amortized time.

Gustedt [9] showed an optimal solution to the union-find problem, in the case when at any time, the actual subsets form disjoint, connected subgraphs of a given planar graph G. In other words, in this problem the allowed unions correspond to the edges of a planar graph and the execution of a union operation can be seen as a contraction of the respective edge.

Our Techniques. It is relatively easy to give a simple *vertex merging data structure* for general graphs, that would process any sequence of contractions in $O(m \log^2 n)$ total time and support the same queries as our data structure in $O(\log n)$ time. To this end, one can store the lists $N(v)$ of neighbors of individual vertices as balanced binary trees. Upon a contraction of an edge uv, or a more general operation of merging two (not necessarily adjacent) vertices u, v, $N(u)$ and $N(v)$ are merged by inserting the smaller set into the larger one (and detecting loops and parallel edges by the way, at no additional cost). If we used hash tables instead of balanced BSTs, we could achieve $O(\log n)$ expected amortized update time and $O(1)$ query time. In fact, such an approach was used in [7].

To obtain the speed-up we take advantage of planarity. Our general idea is to partition the graph into small pieces and use the above simple-minded vertex merging data structures to solve our problem separately for each of the pieces and for the subgraph induced by the vertices contained in multiple pieces (the so-called boundary vertices). Due to the nature of edge contractions, we need to specify how the partition evolves when our graph changes.

The data structure builds an r-division (see Section 2) $\mathcal{R} = P_1, P_2, \ldots$ of G_0 for $r = \log^4 n$. The set $\partial\mathcal{R}$ of boundary vertices (i.e., those shared among at least two pieces) has size $O(n/\log^2 n)$. Let (V_0, E_0) denote the original graph, and (V, E) denote the current graph (after performing some number of contractions). Then we can denote by $\phi : V_0 \to V$ a function such that the initial vertex $v_0 \in V_0$ is contracted into $\phi(v_0)$. We use vertex merging data structures to detect parallel edges and self-loops in the "top-level" subgraph $G[\phi(\partial\mathcal{R})]$, which contains only edges between boundary vertices, and separately for the "bottom-level" subgraphs $G[\phi(V(P_i))] \setminus G[\phi(\mathcal{R})]$. At any time, each edge of G is contained in exactly one of the defined subgraphs, and thus, the distribution of responsibility for handling individual edges is based solely on the initial r-division.

However, such an assignment of responsibilities gives rise to additional difficulties. First, a contraction of an edge in a lower-level subgraph might cause some edges "flow" from this subgraph to the top-level subgraph (i.e., we may get new edges connecting boundary vertices). As such an operation turns out to be costly in our implementation, we need to prove that the number of such events is only $O(n/\log^2 n)$.

Another difficulty lies in the need of keeping the individual data structures synchronized: when an edge of the top-level subgraph is contracted, pairs of vertices in multiple lower-level subgraphs might need to be merged. We cannot afford iterating through all the lower-level subgraphs after each contraction in $G[\phi(\partial\mathcal{R})]$. This problem is solved by maintaining a system of pointers between representations of the same vertex of V in different data structures and another clever application of the smaller-to-larger merge strategy.

Such a two-level data structure would yield a data structure with $O(n \log\log n)$ total update time. To obtain a linear time data structure, we further partition the pieces P_i and add another layer of maintained subgraphs on $O(\log^4 \log^4 n) = O(\log^4 \log n)$ vertices. These subgraphs are so small that we can precompute in $O(n)$ time the self-loops and parallel edges for every possible graph on t vertices and every possible sequence of edge contractions.

We note that this overall idea of recursively reducing a problem with an r-division to a size when microencoding can be used has been previously exploited in [9] and [14] (Gustedt [9] did not use r-divisions, but his concept of a *patching* could be replaced with an r-division). Our data structure can be also seen as a solution to a more general version of the planar union-find problem studied by Gustedt [9]. However, maintaining the status of each edge e of the initial graph G (i.e., whether e has become a self-loop or a parallel edge) subject to edge contractions turns out to be a serious technical challenge. For example, in [9], the requirements posed on the bottom-level union-find data structures are in a sense relaxed and it is not necessary for those to be synchronized with the top-level union-find data structure.

Organization of the Paper. The remaining part of this paper is organized as follows. In Section 2, we introduce the needed notation and definitions, whereas in Section 3 we define the operations that our data structure supports. Then, in Section 4 we present a series of applications of our data structure. In Section 5, we provide a detailed implementation of our data structure. Due to space constraints, many of the proofs, along with the pseudocode for example algorithms using our data structure, can be found in the full version of this paper [10].

2 Preliminaries

Throughout the paper we use the term *graph* to denote an undirected *multigraph*, that is we allow the graphs to have parallel edges and self-loops. Formally, each edge e of such a graph is a pair $(\{u, w\}, \mathrm{id}(e))$ consisting of a pair of vertices and a unique identifier used to distinguish between the parallel edges. For simplicity, we skip this third coordinate and use just uw to denote one of the edges connecting vertices u and w. If the graph contains no parallel edges and no self-loops, we call it *simple*.

For any graph G, we denote by $V(G)$ and $E(G)$ the sets of vertices and edges of G, respectively. A graph G' is called a subgraph of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. We define $G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2))$ and $G_1 \setminus G_2 = (V(G_1), E(G_1) \setminus E(G_2))$. For $S \subseteq V(G)$, we denote by $G[S]$ the *induced subgraph* $(S, \{uv : uv \in E(G), \{u, v\} \subseteq S\})$.

For a vertex $v \in V$, we define $N(v) = \{u : uv \in E, u \neq v\}$ to be the *neighbor set* of v.

A *cycle* of a graph G is a nonempty set $C \subseteq E(G)$, such that for some ordering of edges $C = \{u_1w_1, \ldots, u_kw_k\}$, we have $w_i = u_{i+1}$ for $1 \leq i < k$ and $w_k = u_1$, and the vertices $u_1, \ldots, u_k$ are distinct. The *length* of a cycle C is simply $|C|$. Note that this definition allows cycles of length 1 (self-loop) or 2 (a pair of parallel edges), but does not allow non-simple cycles of length 3 or more. A *cut* is a minimal (w.r.t. inclusion) set $C \subseteq E(G)$, such that $G \setminus C$ has more connected components than G.

Let $G = (V, E)$ be a graph and $xy = e \in E$. We use $G - e$ to denote the graph obtained from G by removing e and G/e to denote the graph obtained by contracting an edge e (in the case of a contraction e may not be a self-loop, i.e., $x \neq y$). We will often look at contraction from the following perspective: as a result of contracting e, all edge endpoints equal to x or y are replaced with some new vertex z. In some cases it is convenient to assume $z \in \{x, y\}$. This yields a 1-to-1 correspondence between the edges of $G - e$ and the edges of G/e. Formally, we assume that the contraction preserves the edge identifiers, i.e., $e_1 \in E(G - e)$ and $e_2 \in E(G/e)$ are corresponding if and only if $\mathrm{id}(e_1) = \mathrm{id}(e_2)$.

Note that contracting an edge may introduce parallel edges and self-loops. Namely, for each edge that is parallel to e in G, there is a self-loop in G/e. And for each cycle of length 3 that contains e in G, there is a pair of parallel edges in G/e.

Planar graphs. An embedding of a planar graph is a mapping of its vertices to distinct points and edges to non-crossing curves in the plane. We say that a planar graph G is *plane*, if some embedding of G is assumed. A face of a connected plane G is a maximal open connected set of points not in the image of any vertex or edge in the embedding of G.

Duality. Let G be a plane graph. We denote by G^* the dual graph of G. Each edge of G naturally corresponds to an edge of G^*. We denote by e^* the edge of G^* that corresponds to $e \in E(G)$. More generally, if $E_1 \subseteq E(G)$ is a set of edges of G, we set $E_1^* = \{e^* | e \in E_1\}$.

We exploit the following relations between G and G^*. Deleting an edge e of G corresponds to contracting the edge e^* in G^*, that is $(G - e)^* = G^*/e^*$. Moreover, $C \subseteq E$ is a cut in G iff C^* is a cycle in G^*. In particular, a bridge e in G corresponds to a self-loop in G^* and a two-edge cut in G corresponds to a pair of parallel edges in G^*.

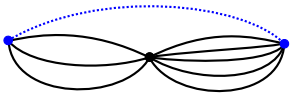

Figure 1 Contracting the blue dotted edge will merge two groups of parallel edges.

Planar graph partitions. Let G be a simple planar graph. Let a *piece* be subgraph of G with no isolated vertices. For a piece P, we denote by ∂P the set of vertices $v \in V(P)$ such that v is adjacent to some edge of G that is not contained in P. ∂P is also called the set of *boundary vertices* of P. An r-division $\mathcal{R}$ of G is a partition of G into $O(n/r)$ edge-disjoint pieces such that each piece $P \in \mathcal{R}$ has $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices. For an r-division $\mathcal{R}$, we also denote by $\partial\mathcal{R}$ the set $\bigcup_{P_i \in \mathcal{R}} \partial P_i$. Clearly, $|\partial\mathcal{R}| = O(n/\sqrt{r})$.

▶ **Lemma 1** ([8, 13, 19]). *An r-division of a planar graph G can be computed in linear time.*

3 The Data Structure Interface

In this section we specify the set of operations that our data structure supports so that it fits our applications. It proves beneficial to look at the graph undergoing contractions from two perspectives.

1. The *adjacency viewpoint* allows us to track the neighbor sets of the individual vertices, as if G was simple at all times.
2. The *edge status viewpoint* allows us to track, for all the original edges E_0, whether they became self-loops or parallel edges, and also track how E_0 is partitioned into classes of pairwise-parallel edges.

Let $G_0 = (V_0, E_0)$ be a planar graph used to initialize the data structure. Recall that any contraction alters both the set of vertices and the set of edges of the graph. Throughout, we let $G = (V, E)$ denote the *current* version of the graph, unless otherwise stated.

Each edge $e \in E(G)$ can be either a self-loop, an edge parallel to some other edge $e' \neq e$ (we call such an edge *parallel*), or an edge that is not parallel to any other edge of G (we call it *simple* in this case). An edge $e \in E(G)$ that is simple might either get contracted or might change into a parallel edge as a result of contracting other edges. Similarly, a parallel edge might either get contracted or might change into a self-loop. Note that, during contractions, neither can a parallel edge ever become simple, nor can a self-loop become parallel.

Observe that parallelism is an equivalence relation on the edges of G. Once two edges e_1, e_2 connecting vertices $u, v \in V$ become parallel, they stay parallel until some edge e_3 (possibly equal to e_1 or e_2) parallel to both of them gets contracted. However, groups of parallel edges might merge (Figure 1) and this might also be a valuable piece of information.

To succinctly describe how the groups of parallel edges change, we report parallelism in a directed manner, as follows. Each group $Y \subseteq E$ of parallel edges in G is assumed to have its *representative* edge $\alpha(Y)$. For $e \in Y$ we define $\alpha(e) = \alpha(Y)$. When two groups of parallel edges $Y_1, Y_2 \subseteq E$ merge as a result of a contraction, the data structure chooses $\alpha(Y_i)$ for some $i \in \{1, 2\}$ to be the new representative of the group $Y_1 \cup Y_2$ and reports an ordered pair $\alpha(Y_{3-i}) \to \alpha(Y_i)$ to the user. We call each such pair a *directed parallelism*. After such an event, $\alpha(Y_{3-i})$ will not be reported as a part of a directed parallelism anymore. The choice of i can also be made according to some fixed strategy, e.g., if the edges are assigned weights $\ell(\cdot)$ then we may choose $\alpha(Y_i)$ so that $\ell(\alpha(Y_i)) \leq \ell(\alpha(Y_{3-i}))$. This is convenient in what Klein and Mozes [12] call *strict optimization problems*, such as MST, where we can discard one of any two parallel edges based only on these edges.

Note that at any point of time the set of directed parallelisms reported so far can be seen as a forest of rooted trees $\mathcal{T}$, such that each tree T of $\mathcal{T}$ represents a group Y of parallel edges of G. The root of T is equal to $\alpha(Y)$.

When some edge is contracted, all edges parallel to it are reported as self-loops. Clearly, each edge e is reported as a self-loop at most once. Moreover, it is reported as a part of a directed parallelism $e \to e'$, $e' \neq e$, at most once.

We are now ready to define the complete interface of our data structure.

- `init`$(G_0 = (V_0, E_0), \ell)$: initialize the data structure. ℓ is an optional weight function.
- $(s, P, L) :=$ `contract`(e), for $e \in E$: contract the edge e. Let $e = uv$. The call `contract`(e) returns a vertex s resulting from merging u and v, and two lists P, L of new directed parallelisms and self-loops, respectively, reported as a result of contraction of e.
- `vertices`(e), for $e \in E$: return $u, v \in V$ such that $e = uv$.
- `neighbors`(u), for $u \in V$: return an iterator to the list $\{(v, \alpha(uv)) : v \in N(u)\}$.
- `deg`(u), for $u \in V$: find the number of neighbors of u in G.
- `edge`(u, v), for $u, v \in V$: if $uv \in E$, then return $\alpha(uv)$. Otherwise, return **nil**.

The following theorem summarizes the performance of our data structure.

▶ **Theorem 2.** *Let $G = (V, E)$ be a planar graph with $|V| = n$ and $|E| = m$. There exists a data structure supporting* `edge`*,* `vertices`*,* `neighbors` *and* `deg` *in $O(1)$ worst-case time, and whose initialization and any sequence of* `contract` *operations take $O(n + m)$ expected time, or $O(n + m)$ worst-case time, if no* `edge` *operations are performed. The data structure supports iterating through the neighbor list of a vertex with $O(1)$ overhead per element.*

4 Applications

Decremental Edge- and Vertex-Connectivity. In the *decremental k-edge (k-vertex) connectivity* problem, the goal is to design a data structure that supports queries about the existence of k edge-disjoint (vertex-disjoint) paths between a pair of given vertices, subject to edge deletions. We obtain improved algorithms for decremental 2-edge-, 2-vertex- and 3-edge-connectivity in dynamic planar graphs. For decremental 2-edge-connectivity we obtain an optimal data structure with both updates and queries supported in amortized $O(1)$ time. In the case of 2-vertex- and 3-edge-connectivity, we achieve the amortized update time of $O(\log n)$, whereas the query time is constant. For all these problems, we improve upon the 20-year-old update bounds by Giammarresi and Italiano [7] by a factor of $O(\log n)$.

▶ **Theorem 3.** *Let $G = (V, E)$ be a planar graph and let $n = |V|$. There exists a deterministic data structure that maintains G subject to edge deletions and can answer 2-edge connectivity queries in $O(1)$ time. Its total update time is $O(n)$.*

Proof. Denote by G_0 the initial graph. Suppose wlog. that G_0 is connected. Let $B(G)$ be the set of all bridges of G. Note that two vertices u, v are in the same 2-edge-connected component of G iff they are in the same connected component of the graph $(V, E \setminus B(G))$.

Observe that if e is a bridge, then deleting e from G does not influence the 2-edge-components of G. Hence, when a bridge e is deleted, we may ignore this deletion. We denote by G' be the graph obtained from G_0 by the same sequence of deletions as G, but ignoring the bridge deletions. This way, G' is connected at all times and the 2-edge-connected components of G' and G are the same. It is also easy to see that $E(G) \setminus B(G) = E(G') \setminus B(G')$ and $B(G) = B(G') \cap E(G)$. Moreover, the set $E(G')$ shrinks in time whereas $B(G')$ only grows.

First we show how the set $B(G')$ is maintained. Recall that $e \in E(G')$ is a bridge of G' iff e^* is a self-loop of G'^*. We build the data structure of Theorem 2 for G'^*, which initially equals G_0^*. As deleting a non-bridge edge e of G' translates to a contraction of a non-loop edge e^* in G'^*, we can maintain $B(G')$ in $O(n)$ total time by detecting self-loops in G'^*.

Denote by H the graph $(V, E(G') \setminus B(G'))$. To support 2-edge connectivity queries, we maintain the graph H with the decremental connectivity data structure of Łącki and Sankowski [14]. This data structure maintains a planar graph subject to edge deletions in linear total time and supports connectivity queries in $O(1)$ time. When an edge e is deleted from G, we first check whether it is a bridge and if so, we do nothing. If e is not a bridge, the set $E(G')$ shrinks and thus we remove the edge e from H. The deletion of e might cause the set $B(G')$ to grow. Any new edge of $B(G')$ is also removed from H afterwards.

To conclude, note that each 2-edge connectivity query on G translates to a single connectivity query in H. All the maintained data structures have $O(n)$ total update time. ◀

As an almost immediate consequence of Theorem 3 we improve upon [6] and obtain an optimal algorithm for the *unique perfect matching* problem when restricted to planar graphs.

▶ **Corollary 4.** *Given a planar graph $G = (V, E)$ with $n = |V|$, in $O(n)$ time we can find a unique perfect matching of G or detect that the number of perfect matchings in G is not 1.*

To obtain improved bounds for 2-vertex connectivity and 3-edge connectivity we use the data structure of Theorem 2 to remove bottlenecks in the existing algorithms by Giammarresi and Italiano [7].

▶ **Theorem 5.** *Let $G = (V, E)$ be a planar graph and let $n = |V|$. There exists a deterministic data structure that maintains G subject to edge deletions and can answer 2-vertex connectivity and 3-edge connectivity queries in $O(1)$ time. Its total update time is $O(n \log n)$.*

Maximal 3-Edge-Connected Subgraphs. A k-edge-connected component of a graph G is a maximal (w.r.t. inclusion) subset S of vertices, such that each pair of vertices in S is k-edge-connected. However, if $k \geq 3$, in the subgraph of G induced by S, some pairs of vertices may not be k-edge-connected (see [2] for an example). Thus, for $k \geq 3$, maximal k-edge-connected subgraphs can be different from k-edge-connected components. Very recently, Chechik et al. [2] showed how to compute maximal k-edge-connected subgraphs in $O((m + n \log n)\sqrt{n})$ time for any constant k, or $O(m\sqrt{n})$ time for $k = 3$. Using the results of [7] one can compute maximal 3-edge-connected subgraphs of a planar multigraph in $O(m + n \log n)$ time. Our new approach allows us to improve this to an optimal $O(m + n)$ time bound.

▶ **Lemma 6.** *The maximal 3-edge-connected subgraphs of a planar graph can be computed in linear time.*

Simple Linear-Time Algorithms. Finally, we present two examples showing that Theorem 2 might be a useful black-box in designing linear time algorithms for planar graphs. The details and the relevant pseudocode can be found in the full version of this paper [10].

▶ **Example 7.** Every planar graph G can be 5-colored in expected linear time.

Proof. A textbook proof of the 5-color theorem proceeds by induction as follows (see Figure 2). Each simple planar graph has a vertex u of degree at most 5. The case when u has degree less than 5 is easy: for any $v \in N(u)$, we can color G/uv inductively, uncontract the edge uv and finally recolor u with a color not used among the vertices $N(u)$. When, however, u has

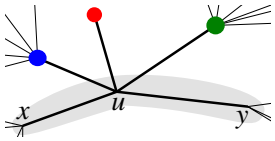

Figure 2 The degree ≤ 5 vertex and its two independent neighbors may be colored using the remaining two colors.

degree exactly 5, there exist two neighbors of x, y of u such that x and y are not adjacent, as otherwise G would contain K_5. We could thus obtain a planar graph G' by contracting both ux and uy. After inductively coloring G' and "uncontracting" ux and uy, we obtain a coloring of G that is valid, except that x, y and u have the same colors assigned. Thus, at most 4 colors are used among the neighbors of u and we recolor u to the remaining color in order to get a valid coloring of G.

Note that this proof can be almost literally converted into a linear time 5-coloring algorithm (see the full version [10] for the pseudocode) using the data structure of Theorem 2 built for G. We only need to maintain a subset Q of vertices of G with degree at most 5. The subset Q can be easily maintained in linear total time, since all vertices that potentially change their degrees after the call `contract`(e) are endpoints of the reported parallel edges. ◀

▶ **Example 8.** An MST of a planar graph G can be computed in linear time.

5 Maintaining a Planar Graph Under Contractions

In this section we prove Theorem 2. We defer the discussion on supporting arbitrary weights $\ell(\cdot)$ to the full version [10]. Hence, in the following, we assume all edges have equal weights.

5.1 A Vertex Merging Data Structure

We first consider a more general problem, which we call the *bordered vertex merging* problem. The data structure presented below will constitute a basic building block of the multi-level data structure. Let us now describe the data structure for the bordered vertex merging problem in detail. Suppose we have a dynamic *simple* planar graph $G = (V, E)$ and a *border set* $B \subseteq V$. Assume G is initially equal to $G_0 = (V_0, E_0)$ and no edge of E_0 connects two vertices of B. The data structure handles the following update operations.

- Merge (or in other words, an identification) of two vertices $u, v \in V$ $(u \neq v)$, such that the graph is still planar. If $\{u, v\} \not\subseteq B$, then u and v have to be connected by an edge and in such a case the merge is equivalent to a contraction of uv.
- Insertion of an edge $e = uv$ (where $uv \notin E$ is not required), preserving planarity.

After each update operation the data structure reports the parallel edges and self-loops that emerge. Once reported, each set of parallel edges is merged into one representative edge. Moreover, the data structure reports and removes any edges that have both endpoints in B. Thus, the following invariants are satisfied before the first and after each modification:

1. G is planar and simple.
2. No edge of E has both its endpoints in B.

Clearly, merging vertices alters the set V by replacing two vertices u, v with a single vertex. Thus, at each step, each vertex of G corresponds to a set of vertices of the initial graph G_0. We explicitly maintain a mapping $\phi : V_0 \to V$ such that for $a \in V_0$, $\phi(a)$ is a

vertex of the current vertex set V "containing" a. The reverse mapping $\phi^{-1} : V \to 2^{V_0}$ is also stored explicitly. We now define how the merge of u and v influences the set B. When $\{u, v\} \subseteq B$, the resulting vertex is also in B. When $u \in B, v \notin B$ (or $v \in B, u \notin B$, resp.), the resulting vertex is included in B in place of u (v, resp.). Finally, for $u, v \notin B$, the resulting vertex does not belong to B either.

Let $\tilde{E}$ be the set of inserted edges. At any time, the edges of E constitute a subset of $E_0 \cup \tilde{E}$ in the following sense: for each $e = xy \in E$ there exists an edge $e' = uv \in E_0 \cup \tilde{E}$ such that $\mathrm{id}(e) = \mathrm{id}(e')$, and vertices u and v have been merged into x and y, respectively.

Note that some modifications might break the second invariant: both an edge insertion and a merge might introduce an edge e with both endpoints in B. We call such an edge a *border edge*. Each border edge e that is not a self-loop is reported and deleted from (or not inserted to) G. Apart from reporting and removing new edges of $B \times B$ appearing in E, we also report the newly created parallel edges that might arise after the modification and remove them. The reporting of parallel edges is done in the form of directed parallelisms, as described in Section 3. Again, it is easy to see that each edge of $E_0 \cup \tilde{E}$ is reported as the first coordinate of a directed parallelism at most once.

Note that an edge e may be first reported parallel (in a directed parallelism of the form $e' \to e$, where $e' \neq e$) and then reported border.

The Graph Representation. The data structure for the bordered vertex merging problem internally maintains G using the data structure of the following lemma for planar graphs.

▶ **Lemma 9** ([1]). *There exists a deterministic, linear-space data structure, initialized in $O(n)$ time, and maintaining a dynamic, simple planar graph H with n vertices, so that:*

- *adjacency queries in H can be performed in $O(1)$ worst-case time,*
- *edge insertions and deletions can be performed in $O(\log n)$ amortized time.*

▶ **Fact 10.** *The data structure of Lemma 9 can be easily extended so that:*

- *Doubly-linked lists $N(v)$ of neighbors, for $v \in V$, are maintained within the same bounds.*
- *For each edge xy of H, some auxiliary data associated with e can be accessed and updated in $O(1)$ worst-case time.*

In addition to the data structure of Lemma 9 representing G, for each unordered pair x, y of vertices adjacent in G, we maintain an edge $\alpha(x, y) = e$, where e is the unique edge in E connecting x and y. Recall that in fact $\alpha(x, y)$ corresponds to some of the original edges of E_0 or one of the inserted edges $\tilde{E}$. By Fact 10, we can access $\alpha(x, y)$ in constant time.

The mapping ϕ is stored in an array, whereas the sets $\phi^{-1}(\cdot)$ – in doubly-linked lists.

Suppose we merge two vertices $u, v \in V$. Instead of creating a new vertex w, we merge one of these vertices into the other. Suppose we merge u into v. In terms of the operations supported by the data structure of Lemma 9, we need to remove each edge ux and insert an edge vx, unless v has been adjacent to x before.

To update our representation, we only need to perform the following steps:

- For each $v_0 \in \phi^{-1}(u)$, set $\phi(v_0) = v$ and add v_0 to $\phi^{-1}(v)$.
- Compute the list $N_u = \{(x, \alpha(u, x)) : x \in N(u)\}$. Remove all edges adjacent to u from G. For each $(x, \alpha(u, x)) \in N_u$, $x \neq v$, check whether $x \in N(v)$ (this can be done in $O(1)$ time, by Lemma 9). If so, report the parallelism $\alpha(u, x) \to \alpha(v, x)$. Otherwise, if vx is not a border edge, insert an edge vx to G and set $\alpha(v, x) = \alpha(u, x)$. If, on the other hand, $v \in B$ and $x \in B$ (i.e., vx is a border edge), report $\alpha(u, x)$ as a border edge.

Observe that our order of updates issued to G guarantees that G remains planar at all times.

The decision whether we merge u into v or v into u heavily affects both the correctness and efficiency of the data structure. First, if one of u, v (say v) is contained in B, whereas the other (say u) is not, we merge u into v. If, however, we have $\{u, v\} \subseteq B$ or $\{u, v\} \subseteq V \setminus B$, we pick a vertex (say u) with a smaller set $\phi^{-1}(u)$ and merge u into v.

To handle an insertion of a new edge $e = xy$, we first check whether xy is a border edge. If so, we discard e and report it. Otherwise, check whether x and y are adjacent in G. If so, report the parallelism $e \to \alpha(x, y)$. If not, add an edge xy to G and set $\alpha(x, y) = e$.

▶ **Lemma 11.** *Let G be a graph initially equal to a simple planar graph $G_0 = (V_0, E_0)$ such that $n = |V_0|$. There is a data structure for the bordered vertex merging problem that processes any sequence of modifications of G_0, along with reporting parallelisms and border edges, in $O((n + f) \log^2 n + m)$ total time, where m is the total number of edge insertions and f is the total number of insertions of edges connecting non-adjacent vertices.*

Proof. Clearly, by Lemma 9, building the initial representation takes $O(n \log n)$ time, as we insert $O(n)$ edges to G. The reporting of parallel edges and border edges takes $O(n + m)$ time, since each (initial or inserted) edge is reported as a border edge or occurs as the first coordinate of a reported directed parallelism at most once.

Also note that, by Lemma 9, an insertion of a parallel edge costs $O(1)$ time, for a total of $O(m)$ time over all insertions, as G is not updated in that case. Recall that, by Fact 10, accessing and updating values $\alpha(x, y)$ for $xy \in E(G)$ takes $O(1)$ time.

The total cost of maintaining the representation of G is $O(g \log n)$, where g is the total number of edge updates to the data structure of Lemma 9. We prove that $g = O((n+f) \log n)$. To this end, we look at the merge of u into v from a different perspective: instead of removing an edge $e = ux$ and inserting an edge vx, imagine that we simply change an endpoint u of e to v, but the edge itself does not lose its identity. Then, new edges in G are only created either during the initialization or by inserting an edge connecting the vertices that have not been previously adjacent in G. Hence, there are $O(n + f)$ creations of new edges.

Consider some edge $e = xy$ of G immediately after its creation. Denote by $q(e)$ the pair $(|\phi^{-1}(x)|, |\phi^{-1}(y)|)$. The value of $q(e)$ always changes when some endpoint of e is updated. Suppose a merge of u into v ($u \neq v$) causes the change of some endpoint u of e to v. We either we have $u \notin B$ and $v \in B$ or $|\phi^{-1}(v)| \geq |\phi^{-1}(u)|$ before the merge. The former situation can arise at most once per each endpoint of e, since we always merge a non-border vertex into a border vertex, if such case arises. In the latter case, on the other hand, one coordinate of $q(e)$ grows at least by a factor of 2, and clearly this can happen at most $O(\log n)$ times, as the size of any $\phi^{-1}(x)$ is never more than n. Since there are $O(n + f)$ "created" edges, and each such edge undergoes $O(\log n)$ endpoint updates, indeed we have $g = O((n + f) \log n)$.

A very similar argument can be used to show that the total time needed to maintain the mapping ϕ along with the reverse mapping ϕ^{-1} is $O(n \log n)$. ◀

A Micro Data Structure. In order to obtain an optimal data structure, we need the following specialized version of the bordered vertex merging data structure that handles very small graphs in linear total time. Suppose we disallow inserting new edges into G. Additionally, assume we are allowed to perform some preprocessing in time $O(n)$. Then, due to a monotonous nature of allowed operations on G, when the size of G_0 is very small compared to n, we can maintain G faster than by using the data structure of Lemma 11.

▶ **Lemma 12.** *After preprocessing in $O(n)$ time, we can repeatedly solve the bordered vertex merging problem without edge insertions for planar simple graphs G_0 with $t = O(\log^4 \log^4 n)$ vertices in $O(t)$ time.*

5.2 A Multi-Level Data Structure

Recall that our goal is to maintain G under contractions. Below we describe in detail how to take advantage of graph partitioning and bordered vertex merging data structures to obtain a linear time solution. To simplify the further presentation, we assume that the initial version $G_0 = (V_0, E_0)$ of G is simple and of constant degree. The standard reduction assuring that is described in the full version [10].

We build an r-division $\mathcal{R} = \{P_1, P_2, \ldots, \}$ of G with $r = \log^4 n$, where $n = |V_0|$ (see Lemma 1). Then, for each piece $P_i \in \mathcal{R}$, we build an r-division $\mathcal{R}_i = \{P_{i,1}, P_{i,2}, \ldots\}$ of P_i with $r = \log^4 \log^4 n$. By Lemma 1, building all the necessary pieces takes $O(n)$ time in total. Since G_0 is of constant degree, any vertex $v \in V_0$ is contained in $O(1)$ pieces of $\mathcal{R}$. Analogously, for any $v \in P_i$, v is contained in $O(1)$ pieces of $\mathcal{R}_i$.

As G undergoes contractions, let $\phi : V_0 \to V$ be a mapping such that for each $v \in V_0$, v "has been merged" into $\phi(v)$. As we later describe, a vertex resulting from contracting an edge uv will be called either u or v, which guarantees that $V \subseteq V_0$ at all times. Of course, initially $\phi(v) = v$ for each $v \in V = V_0$.

Let $\overline{G} = (V, \overline{E})$ denote the maximal simple subgraph of G, i.e., the graph G with self-loops discarded and each group Y of parallel edges replaced with a single edge $\alpha(Y)$. The key component of our data structure is a 3-level set of (possibly micro-) bordered vertex merging data structures $\Pi = \{\pi\} \cup \{\pi_i : P_i \in \mathcal{R}\} \cup \{\pi_{i,j} : P_i \in \mathcal{R}, P_{i,j} \in \mathcal{R}_i\}$. The data structures Π form a tree such that π is the root, $\{\pi_i : P_i \in \mathcal{R}\}$ are the children of π and $\{\pi_{i,j} : P_{i,j} \in \mathcal{R}_i\}$ are the children of π_i. For $\mathcal{D} \in \Pi$, let $par(\mathcal{D})$ be the parent of $\mathcal{D}$ and let $A(\mathcal{D})$ be the set of ancestors of $\mathcal{D}$. We call the value $h(\mathcal{D}) = |A(\mathcal{D})|$ a *level* of $\mathcal{D}$. The data structures of levels 0 and 1 are stored as data structures of Lemma 11, whereas the data structures of level 2 are stored as micro structures of Lemma 12.

Each data structure $\mathcal{D} \in \Pi$ has a defined set $V_\mathcal{D} \subseteq V_0$ of *interesting vertices*, defined as follows: $V_\pi = \partial\mathcal{R}$, $V_{\pi_i} = \partial P_i \cup \partial\mathcal{R}_i$ and $V_{\pi_{i,j}} = V(P_{i,j})$. The data structure $\mathcal{D}$ maintains a certain subgraph $G_\mathcal{D}$ of $\overline{G}$ defined inductively as follows (recall that we define $G_1 \setminus G_2$ to be a graph containing all vertices of G_1 and edges of G_1 that do not belong to G_2)

$$G_\mathcal{D} = \overline{G}[\phi(V_\mathcal{D})] \setminus \left(\bigcup_{\mathcal{D}' \in A(\mathcal{D})} G_{\mathcal{D}'} \right).$$

▶ **Fact 13.** *For any $\mathcal{D} \in \Pi$, $G_\mathcal{D}$ is a minor of G_0.*

▶ **Fact 14.** *For any $uv = e \in \overline{E}$, there exists $\mathcal{D} \in \Pi$ such that $e \in E(G_\mathcal{D})$.*

Each $\mathcal{D} \in \Pi$ is initialized with the graph $G_\mathcal{D}$, according to the initial mapping $\phi(v) = v$ for any $v \in V_0$. We define the set of *ancestor vertices* $AV_\mathcal{D} = V_\mathcal{D} \cap \left(\bigcup_{\mathcal{D}' \in A(\mathcal{D})} V_{\mathcal{D}'}\right)$.

Now we discuss what it means for the bordered vertex merging data structure $\mathcal{D}$ to maintain the graph $G_\mathcal{D}$. Note that the vertex set used to initialize $\mathcal{D}$ is $V_\mathcal{D}$. We write $\phi_\mathcal{D}, \phi_\mathcal{D}^{-1}$ to denote the mappings ϕ, ϕ^{-1} maintained by $\mathcal{D} \in \Pi$, respectively. Throughout a sequence of contractions, we maintain the following invariants for any $\mathcal{D} \in \Pi$:

- There is a 1-1 mapping between the sets $\phi(V_\mathcal{D})$ and $\phi_\mathcal{D}(V_\mathcal{D})$ such that for the corresponding vertices $x \in \phi(V_\mathcal{D})$ and $y \in \phi_\mathcal{D}(V_\mathcal{D})$ we have $\phi_\mathcal{D}^{-1}(y) = \phi^{-1}(x) \cap V_\mathcal{D}$. We also say that x is *represented* in $\mathcal{D}$ in this case.
- There is an edge $xy \in E(G_\mathcal{D})$ if and only if there is an edge $x'y'$ in the graph maintained by $\mathcal{D}$, where $x', y' \in \phi_\mathcal{D}(V_\mathcal{D})$ are the corresponding vertices of x and y, respectively.
- The border set $B_\mathcal{D}$ of $\mathcal{D}$ is always equal to $\phi_\mathcal{D}(AV_\mathcal{D})$.

Thus, the graph maintained by $\mathcal{D}$ is isomorphic to $G_{\mathcal{D}}$ but can technically use a different vertex set. Observe that in $G_{\mathcal{D}}$ there are no edges between the vertices $\phi(AV_{\mathcal{D}})$ and the following fact describes how this is reflected in $\mathcal{D}$.

▶ **Fact 15.** *In the graph stored in $\mathcal{D}$, no two vertices of $B_{\mathcal{D}}$ are adjacent.*

Note that as the sets $V_{\mathcal{D}}$ and $V_{\mathcal{D}'}$ might overlap for $\mathcal{D} \neq \mathcal{D}'$, the vertices of V can be represented in multiple data structures.

▶ **Lemma 16.** *Suppose for $v \in V$ we have $v \in V(G_{\mathcal{D}_1})$ and $v \in V(G_{\mathcal{D}_2})$. Then, $v \in V(G_{\mathcal{D}})$, where $\mathcal{D}$ is the lowest common ancestor of $\mathcal{D}_1$ and $\mathcal{D}_2$.*

By Lemma 16, each vertex $v \in V$ is represented in a unique data structure of minimal level, a lowest common ancestor of all data structures where v is represented. We denote such a data structure by $\mathcal{D}(v)$. Observe that for any $\mathcal{D} \in \Pi$ the vertices $\{v : \mathcal{D}(v) = \mathcal{D}\}$ are represented in $\mathcal{D}$ by $\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus \phi_{\mathcal{D}}(AV_{\mathcal{D}})$.

We now describe the way we index the vertices of V. This is required, as upon a contraction, our data structure returns an identifier of a new vertex. We also reuse the names of the initial vertices V_0, as the bordered vertex merging data structures do. Namely, a vertex $v \in V$ is labeled with $\phi_{\mathcal{D}(v)}(v') \in V_0$, where v' represents v in $\mathcal{D}(v)$.

Note that, as the bordered vertex merging data structures always merge one vertex involved into the other, for any $\mathcal{D} \in \Pi$ we have $\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus \phi_{\mathcal{D}}(AV_{\mathcal{D}}) \subseteq V_{\mathcal{D}} \setminus AV_{\mathcal{D}}$. Hence the label sets used by distinct sets $\{v : \mathcal{D}(v) = \mathcal{D}\}$ are distinct, since the sets of the form $V_{\mathcal{D}} \setminus AV_{\mathcal{D}}$ are pairwise disjoint. Such a labeling scheme makes it easy to find the data structure $\mathcal{D}(v)$ by looking only at the label.

For brevity, in the following we sometimes do not distinguish between the set V and the set of labels $\bigcup_{\mathcal{D}\in\Pi} (\phi_{\mathcal{D}}(V_{\mathcal{D}}) \setminus \phi_{\mathcal{D}}(AV_{\mathcal{D}}))$.

▶ **Lemma 17.** *Let $uv = e \in \overline{E}$ and $h(\mathcal{D}(u)) \geq h(\mathcal{D}(v))$. Then $e \in E(G_{\mathcal{D}(u)})$ and either $\mathcal{D}(u) = \mathcal{D}(v)$ or $\mathcal{D}(u)$ is a descendant of $\mathcal{D}(v)$.*

▶ **Lemma 18.** *Let uv be an edge of some $G_{\mathcal{D}}$, $\mathcal{D} \in \Pi$. If $\{u, v\} \subseteq V(G_{\mathcal{D}'})$, where $\mathcal{D}' \neq \mathcal{D}$, then $\mathcal{D}'$ is a descendant of $\mathcal{D}$ and both u and v are represented as border vertices of $\mathcal{D}'$.*

▶ **Lemma 19.** *Let $v \in \phi(V_{\mathcal{D}})$, where $\mathcal{D} \in \Pi$. Then, v is represented in $O(|\phi_{\mathcal{D}}^{-1}(v)|)$ data structures $\mathcal{D}'$ such that $par(\mathcal{D}') = \mathcal{D}$.*

We also use the following auxiliary components for each $\mathcal{D} \in \Pi$:

- For each $x \in \phi_{\mathcal{D}}(AV_{\mathcal{D}})$ we maintain a pointer $\beta_{\mathcal{D}}(x)$ into $y \in \phi_{par(\mathcal{D})}(AV_{\mathcal{D}})$, such that x and y represent the same vertex of the maintained graph G.
- A dictionary (we use a balanced BST) $\gamma_{\mathcal{D}}$ mapping a pair $(\mathcal{D}', x)$, where $\mathcal{D}'$ is a child of $\mathcal{D}$ and $x \in \phi_{\mathcal{D}}(V_{\mathcal{D}})$, to a vertex $y \in \phi_{\mathcal{D}'}(AV_{\mathcal{D}})$ iff x and y represent the same vertex of V.

Another component of our data structure is the forest $\mathcal{T}$ of reported parallelisms: for each reported parallelism $e \to \alpha(e)$, we make e a child of $\alpha(e)$ in $\mathcal{T}$. Note that the forest $\mathcal{T}$ allows us to go through all the edges parallel to $\alpha(e)$ in time linear in their number.

▶ **Lemma 20.** *For $v_0 \in V_0$, we can compute $\phi(v_0)$ and find $\mathcal{D}(\phi(v_0))$ in $O(1)$ time.*

▶ **Lemma 21.** *Let $v \in V(G_{\mathcal{D}})$. For any $\mathcal{D}'$, such that $par(\mathcal{D}') = \mathcal{D}$, we can compute the vertex v' representing v in $G_{\mathcal{D}'}$ (or detect that such v' does not exist) in $O(\log |V_{\mathcal{D}}|)$ time.*

We now describe how to implement the call $(s, P, L) := \texttt{contract}(e)$, where $uv = e \in E$, $u, v \in V$. Suppose the initial endpoints of e were $u_0, v_0 \in V_0$. First, we iterate through the tree $T_e \in \mathcal{T}$ containing e to find $\alpha(e)$. By Lemma 20, we can find the vertices u, v along with the respective data structures $\mathcal{D}(u), \mathcal{D}(v)$, based on u_0, v_0 in $O(1)$ time. Assume wlog. that $h(\mathcal{D}(u)) \geq h(\mathcal{D}(v))$. By Lemma 17, $\alpha(e)$ is an edge of $G_{\mathcal{D}(u)}$. Although we are asked to contract e, we conceptually contract $\alpha(e)$, by issuing a merge of u and v to $\mathcal{D}(u)$. To reflect that we were actually asked to contract e, we include all the edges of $T_e \setminus \{e\}$ in L as self-loops. The merge might make $\mathcal{D}(u)$ report some parallelisms $e_1 \to e_2$. In such a case we report $e_1 \to e_2$ to the user (by including it in P) and update the forest $\mathcal{T}$.

We now have to reflect the contraction of e in all the required data structures $\mathcal{D} \in \Pi$, so that our invariants are satisfied. Assume wlog. that u is merged into v in $\mathcal{D}$. If before the contraction, both u and v were the vertices of some $G_{\mathcal{D}'}$, $\mathcal{D}' \neq \mathcal{D}$, then by Lemma 18, $\mathcal{D}'$ is a descendant of $\mathcal{D}$. By a similar argument as in the proof of Lemma 11, we can afford to iterate through $\phi_{\mathcal{D}}^{-1}(u)$ without increasing the asymptotic performance of the u-into-v merge performed by $\mathcal{D}$, as long as we spend $O(\log |V_{\mathcal{D}}|)$ time per element of $\phi_{\mathcal{D}}^{-1}(u)$. By Lemma 19, there are $O(|\phi_{\mathcal{D}}^{-1}(u)|)$ data structures $\mathcal{D}_1, \mathcal{D}_2, \ldots$ that are the children of $\mathcal{D}$ and contain the representation of u. For each such $\mathcal{D}_i$, we first use the dictionary $\gamma_{\mathcal{D}}$ to find the vertex x representing u in $\mathcal{D}_i$, and update $\beta_{\mathcal{D}_i}(x)$ to v. Then, using Lemma 21, we check whether $v \in V(G_{\mathcal{D}_i})$ in $O(\log |V_{\mathcal{D}}|)$ time. If not, we set $\gamma_{\mathcal{D}}(\mathcal{D}_i, v)$ to x. Otherwise, we merge u and v in $\mathcal{D}_i$ and handle this merge – in terms of updating the auxiliary components β and γ – analogously as for $\mathcal{D}$. This is legal, as $u, v \in \phi_{\mathcal{D}_i}(AV_{\mathcal{D}_i})$ and thus u and v are border vertices in $\mathcal{D}_i$, by Fact 15. The merge may cause $\mathcal{D}_i$ to report some parallelisms. We handle them as described above in the case of the data structure $\mathcal{D}$. Note however that merging border vertices cannot cause reporting of new border edges (i.e., those with both endpoints in $B_{\mathcal{D}_i}$).

The merge of u and v in $\mathcal{D}$ might also create some new edges $e' = xy$ between the vertices $\phi_{\mathcal{D}}(AV_{\mathcal{D}})$ in $G_{\mathcal{D}}$. Note that in this case $\mathcal{D}$ reports xy as a border edge and also we know that $h(\mathcal{D}(x)) < h(\mathcal{D})$ and $h(\mathcal{D}(y)) < h(\mathcal{D})$. Hence, e' should end up in some of the ancestors of $\mathcal{D}$. We insert e' to $par(\mathcal{D})$. $par(\mathcal{D})$ might also report xy as a border edge and in that case e' is inserted to the grandparent of $\mathcal{D}$. It is also possible that e' will be reported a parallel edge in some of the ancestors of $\mathcal{D}$: in such a case an appropriate directed parallelism is added to P.

Note that all the performed merges and edge insertions are only used to make the graphs represented by the data structures satisfy their definitions. Fact 13 implies that the represented graphs remain planar at all times.

We now describe how the other operations are implemented. To compute $u, v \in V$ such that $\{u, v\} = \texttt{vertices}(e)$, where $e \in E$, we first use Lemma 20 to compute $u = \phi(u_0)$ and $v = \phi(v_0)$, where u_0, v_0 are the initial endpoints of e. Clearly, this takes $O(1)$ time.

To maintain the values $\texttt{deg}(v)$ of each $v \in V$, we simply set $\texttt{deg}(s) := \texttt{deg}(u) + \texttt{deg}(v) - 1$ after a call $(s, P, L) := \texttt{contract}(e)$. Additionally, for each directed parallelism $e_1 \to e_2$ we decrease $\texttt{deg}(x)$ and $\texttt{deg}(y)$ by one, where $\{x, y\} = \texttt{vertices}(e_1)$.

For each $u \in V$ we maintain a doubly-linked list $\mathcal{E}(u) = \{\alpha(uv) : uv \in \overline{E}\}$. Additionally, for each $e \in \overline{E}$ we store the pointers to the two occurrences of e in the lists $\mathcal{E}(\cdot)$. Again after a call $(s, P, L) := \texttt{contract}(e)$, where $e = uv$, we set $\mathcal{E}(s)$ to be a concatenation of the lists $\mathcal{E}(u)$ and $\mathcal{E}(v)$. Finally, we remove all the occurrences of edges $\{\alpha(e)\} \cup \{e_1 : (e_1 \to e_2) \in P\}$ from the lists $\mathcal{E}(\cdot)$. Now, the implementation of the iterator $\texttt{neighbors}(u)$ is easy, as the endpoints not equal to u of the edges in $\mathcal{E}(u)$ form exactly the set $N(u)$.

▶ **Lemma 22.** *The operations* `vertices`, `deg` *and* `neighbors` *run in* $O(1)$ *worst-case time.*

To support the operation $\texttt{edge}(u, v)$ in $O(1)$ time, we first turn all the dictionaries $\gamma_{\mathcal{D}}$ into hash tables with $O(1)$ expected update time and $O(1)$ worst-case query time [3]. Our data structure thus ceases to be deterministic, but we obtain a more efficient version of Lemma 21 that allows us to compute the representation of a vertex in a child data structure $\mathcal{D}'$ in $O(1)$ time. By Lemma 17, the edge uv can be contained in either $\mathcal{D}(u)$ or $\mathcal{D}(v)$, whichever has greater level. Wlog. suppose $h(\mathcal{D}(u)) \geq h(\mathcal{D}(v))$. Again, by Lemma 17, $\mathcal{D}(u)$ is a descendant of $\mathcal{D}(v)$. Thus, we can find v in $\mathcal{D}(u)$ by applying Lemma 21 at most twice.

▶ **Lemma 23.** *If the dictionaries $\gamma_{\mathcal{D}}$ are implemented as hash tables, the operation* `edge` *runs in $O(1)$ worst-case time.*

▶ **Lemma 24.** *The cost of all operations on the data structures $\mathcal{D} \in \Pi$ is $O(n)$.*

Proof of Theorem 2. To initialize our data structure, we initialize all the data structures $\mathcal{D} \in \Pi$ and the auxiliary components. This takes $O(n)$ time. The time needed to perform any sequence of operations `contract` is proportional to the total time used by the data structures Π, as the cost of maintaining the auxiliary components can be charged to the operations performed by the individual structures of Π. By Lemma 24, this time is $O(n)$. If the dictionaries $\gamma_{\mathcal{D}}$ are implemented as hash tables, this bound is valid only in expectation.

By combining the above with Lemmas 22 and 23, the theorem follows. ◀

References

1 Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, pages 342–351, 1999. doi:10.1007/3-540-48447-7_34.

2 Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1900–1918, 2017. doi:10.1137/1.9781611974782.124.

3 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994. doi:10.1137/S0097539791194094.

4 Jack Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, pages 449–467, 1965.

5 Greg N. Frederickson. On linear-time algorithms for five-coloring planar graphs. *Inf. Process. Lett.*, 19(5):219–224, 1984. doi:10.1016/0020-0190(84)90056-5.

6 Harold N. Gabow, Haim Kaplan, and Robert Endre Tarjan. Unique maximum matching algorithms. *J. Algorithms*, 40(2):159–183, 2001. Announced at STOC'99. doi:10.1006/jagm.2001.1167.

7 Dora Giammarresi and Giuseppe F. Italiano. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica*, 16(3):263–287, 1996. doi:10.1007/BF01955676.

8 Michael T. Goodrich. Planar separators and parallel polygon triangulation. *J. Comput. Syst. Sci.*, 51(3):374–389, 1995. doi:10.1006/jcss.1995.1076.

9 Jens Gustedt. Efficient union-find for planar graphs and other sparse graph classes. *Theor. Comput. Sci.*, 203(1):123–141, 1998. doi:10.1016/S0304-3975(97)00291-0.

10 Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, Eva Rotenberg, and Piotr Sankowski. Contracting a planar graph efficiently, 2017. arXiv:1706.10228.

11 David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-out algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'93, pages 21–30, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

12 Philip N. Klein and Shay Mozes. Optimization algorithms for planar graphs, 2017. URL: `http://planarity.org`.

13 Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 505–514, 2013. `doi:10.1145/2488608.2488672`.

14 Jakub Łącki and Piotr Sankowski. Optimal decremental connectivity in planar graphs. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, pages 608–621, 2015. `doi:10.4230/LIPIcs.STACS.2015.608`.

15 Martin Mareš. Two linear time algorithms for mst on minor closed graph classes. *Archivum mathematicum*, 40(3):315–320, 2002.

16 Tomomi Matsui. The minimum spanning tree problem on a planar graph. *Discrete Applied Mathematics*, 58(1):91–94, 1995. `doi:10.1016/0166-218X(94)00095-U`.

17 David W. Matula, Yossi Shiloach, and Robert E. Tarjan. Two linear-time algorithms for five-coloring a planar graph. Technical report, Stanford University, Stanford, CA, USA, 1980.

18 Neil Robertson, Daniel P. Sanders, Paul Seymour, and Robin Thomas. Efficiently four-coloring planar graphs. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC'96, pages 571–575, New York, NY, USA, 1996. ACM. `doi:10.1145/237814.238005`.

19 Freek van Walderveen, Norbert Zeh, and Lars Arge. Multiway simple cycle separators and I/O-efficient algorithms for planar graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 901–918, 2013. `doi:10.1137/1.9781611973105.65`.